Défis pour le Génie de la Programmation et du Logiciel
GDR CNRS GPL – http://gdr-gpl.cnrs.fr/

Coordonnés par
Laurence Duchien (LIFL, INRIA, Univ. Lille 1) & Yves Ledru (LIG, Univ. Grenoble-1)

avec la collaboration de

Y. Aït-Ameur (LISI, Poitiers),
P. Albert (IBM),
N. Anquetil (LIFL, INRIA, Univ. Lille 1),
G. Arévalo (LIFIA, UNLP, Argentine),
Z. Azmeh (LIRMM, Univ. Montpellier 2),
M. Blay-Fornarino (I3S, Univ. Nice-Sophia-Antipolis),
P. Castéran (LABRI, U. Bordeaux),
P. Cointe (LINA, INRIA, Ecole des Mines de Nantes),
P. Collet (I3S, Univ. Nice-Sophia-Antipolis),
B. Combemale (IRISA, Univ. Rennes 1),
C. Consel (LABRI, INRIA, Univ. Bordeaux 1),
S. Denier (LIFL, INRIA, Univ. Lille 1),
R. Douence (LINA, INRIA, Ecole des Mines de Nantes),
C. Dubois (CEDRIC, ENSIIE),
S. Ducasse (LIFL, INRIA, Univ. Lille 1),
R. Ducournau (LIRMM, Univ. Montpellier 2),
S. Dupuy-Chessa (LIG, Univ. Grenoble 1),
A. Front (LIG, Univ. Grenoble 2),
R. Giroudeau (LIRMM, Univ. Montpellier 2),
A. Grost (ATOS),
R. Groz (LIG, Grenoble INP),
A. Gotlieb (INRIA),
G. Hains (LACL, Univ. Paris-Est-Créteil),
M. Huchard (LIRMM, Univ. Montpellier 2),
J.-M. Jézéquel (IRISA, Univ. Rennes 1),
J.-C. König (LIRMM, Univ. Montpellier
J. Laval (LIFL, INRIA, Univ. Lille 1),
X. Le Pallec (LIFL, Univ. Lille 1),
A. Lèbre (LINA, INRIA, Ecole des Mines de Nantes),
T. Ledoux (LINA, INRIA, Ecole des Mines de Nantes),
L. Ledrich (ALTEN Nord),
J.-M. Menaud (LINA, INRIA, Ecole des Mines de Nantes),
T. Nodenot (LIUPPA, Univ. de Pau et des Pays de l'Adour),
J. Noyé (LINA, INRIA, Ecole des Mines de Nantes),
A.-M. Pinna-Dery (I3S, Univ. Nice-Sophia-Antipolis),
D. Pollet (LIFL, INRIA, Univ. Lille 1),
M.-L. Potet, (Verimag, Grenoble INP),
C. Tibermacine (LIRMM, Univ. Montpellier 2),
R. Rouvoy (LIFL, INRIA, Univ. Lille1),
S. Rusinek (Psitec),
L. Seinturier (LIFL, INRIA, Univ. Lille1),
D. Seriai (LIRMM, Univ. Montpellier 2),
M. Südholt (LINA, INRIA, Ecole des Mines de Nantes),
C. Urtado (LGI2P, Ecole des Mines d'Alès),
S. Vauttier (LGI2P, Ecole des Mines d'Alès),
S. Vignes (Dpt Informatique et Réseaux, Telecom-ParisTech),
H. Waeselynk (LAAS),
M. Ziane (LIP6, Univ. P. et M. Curie).

2),
P. Lahire (I3S, Univ. Nice-Sophia-
Antipolis),
R. Laleau (LACL, Univ. Paris-Est-Créteil),

Ce document a été élaboré dans le cadre du groupement de recherche CNRS « Génie de la Programmation et du Logiciel » (GDR GPL) sous la direction de Laurence Duchien et de Yves Ledru, avec la collaboration des groupes de travail du GDR.

Après un résumé synthétique des différentes contributions, ce document présente des problématiques qui ont été identifiées comme particulièrement importantes dans les années qui viennent dans les domaines du génie logiciel et de la programmation. Sans faire un recensement exhaustif, le repérage effectué par les groupes de travail du GDR GPL contribue à une cartographie du positionnement français et suggère plusieurs thématiques stratégiques dans ces domaines.

Résumé des contributions

Jean-Marc Jézéquel constate qu'on n'écrit plus des logiciels en partant de zéro mais en faisant évoluer, en continu, des logiciels existants. Dans ce contexte, les concepts actuels de structuration (classes, paquets, etc.) s'avèrent insuffisants et il est nécessaire d'introduire de nouveaux concepts de structuration qui supportent mieux les considérations transverses. Nicolas Anquetil et ses collègues notent que, dans ce contexte d'évolution continue des logiciels, la qualité de la structure des logiciels est déterminante pour faciliter ces migrations. Il s'agit donc d'être capable d'identifier les bonnes structures, et le cas échéant de les remodulariser pour les adapter et les recréer.

Patrick Albert et ses collègues partent du constat largement accepté que l'implication de l'utilisateur dans la construction d'un logiciel est un élément déterminant pour le succès ou l'échec du projet. Leur proposition est de faire participer plus activement l'utilisateur, expert d'un domaine, à l'élaboration des modèles dans une démarche d'Ingénierie Dirigée par les Modèles (IDM). Une autre façon d'impliquer l'utilisateur consiste à lui proposer un langage de description spécifique pour son domaine (DSL). Charles Consel remarque que la définition d'un tel langage est une activité complexe et propose d'étudier les manques des propositions actuelles pour ensuite développer des outils de développement dédiés aux DSL.

Dans ce contexte où la complexité des logiciels se traduit par une activité d'adaptation et d'intégration, Gabriela Arévalo et ses collègues notent le besoin de nouvelles infrastructures pour la recherche de composants et de services. Ces infrastructures devraient aller au delà de la simple recherche pour permettre aux utilisateurs de ces composants et services de partager leurs expériences et connaissances. Lionel Seinturier attire notre attention sur les nombreux défis que posent l'approche service et son prolongement que constitue le "*cloud computing*". Pour le Génie Logiciel, le défi concerne la définition d'un modèle universel de composants et de services qui couvre l'ensemble de leur cycle de vie. Romain Rouvoy anticipe le développement de nouvelles utilisations de l'Internet où chacun aura la possibilité de publier des services virtuels ou physiques. La conception et la publication de ces ressources appellent le développement de nouveaux outils logiciels, adaptés à la diversité de ces objets communicants.

Jean-Marc Menaud et ses collègues constatent que les considérations environnementales nous forcent à considérer les performances des logiciels non seulement en consommation de temps et de mémoire, mais également en termes de consommation énergétique. Il s'agit dès lors de sensibiliser l'utilisateur final et le développeur à ce problème en leur offrant des mécanismes d'introspection/réflexion adaptés. Gaétan Hains va également dans ce sens en proposant de remonter les mécanismes de *monitoring* dans les plates-formes à large échelle du niveau système au niveau de l'intergiciel.

Mikal Ziane remarque que l'une des difficultés du développement logiciel vient du fait que ces outils partagent peu de connaissances avec leurs utilisateurs. Dans les années 80, le projet américain *Knowledge-Based Software Engineering* avait fait le même constat, sans parvenir à le résoudre. Depuis, les connaissances et les outils ont significativement évolué et il est probablement temps de tenter à nouveau l'intégration des outils et des connaissances. Finalement, Laurence Duchien attire notre attention sur le besoin de construire des logiciels flexibles qui s'adaptent à leur contexte d'utilisation. Cela revient à prendre en compte l'incertitude dans toutes les phases du développement logiciel et suppose le développement de modèles de haut niveau pour représenter l'incertitude.

Le groupe MFDL (Méthodes Formelles pour le Développement du Logiciel) pose le défi de l'intégration maîtrisée et raisonnée des méthodes formelles dans le développement de systèmes à logiciel prépondérant. Cela amène notamment aux problèmes suivants : établir une chaîne continue de développement qui va de l'analyse système à la production de code, et garantir la traçabilité sémantique tout au long de cette chaîne. Les difficultés de passage à l'échelle de ces méthodes imposent aussi des choix quant aux parties qui feront l'objet de tels développements et leurs liens avec des parties développées de façon plus classique.

Le groupe MTV2 (Méthodes de Test pour la Validation et la Vérification) identifie trois défis pour la définition de processus et techniques de test mieux étayés, plus fiables en termes de sûreté et de sécurité, et capables de passer à l'échelle. Le premier défi concerne l'intégration du test aux nouvelles pratiques de développement que constituent le génie logiciel à base de modèles, les méthodes agiles et le développement « off-shore ». Le deuxième défi répond à la société qui demande des garanties en termes de fiabilité et de sécurité. Le troisième défi est celui de la diversité et de la complexité des environnements où sont plongés les systèmes à tester, ce qui les rend plus difficilement contrôlables et observables, et fait exploser le nombre de configurations et de comportements à tester.

Sylvie Vignes attire notre attention sur le défi que représente « l'adaptation diffuse », c'est-à-dire la capacité d'un système à modifier les comportements individuels de ses artefacts pour faire émerger un comportement global en interaction avec son environnement. De nombreuses applications relèvent de cette problématique : contrôle d'un essaim de micro-robots, auto-organisation d'objets hétérogènes destinés à la domotique, réseaux de capteurs, etc. Un constituant essentiel de ce défi est l'ingénierie des besoins qui doit à la fois exprimer le

comportement global et les exigences sur les composants individuels, tout en tenant compte du caractère incertain et des capacités d'adaptation du système.

Le groupe LTP (Langages, Types et Preuves) note que la diffusion des méthodes de développement basées sur la preuve reste limitée. Il diagnostique que cet état de fait est lié à la difficulté de prise en main des outils de preuve, à leur capacité à passer à l'échelle et à la nécessité de valider les spécifications et hypothèses de départ. Il propose des pistes pour relever le défi de l'adoption des outils de preuve et propose de faciliter la validation des spécifications par l'utilisation de langages dédiés aux domaines concernés.

1. Introduction

L'omniprésence de l'informatique dans notre quotidien à l'échelle de l'embarqué et de l'intelligence ambiante, l'extension du web au niveau de la planète, mais également dans les objets du quotidien, le développement de grandes infrastructures de calcul ou des centres de traitement de grandes masses de données soulèvent de nombreuses questions pour le génie de la programmation et du logiciel. Parmi ces questions, quelles sont celles qui correspondent à des défis que devront relever les chercheurs dans le domaine du génie de la programmation et du logiciel à échéance de 5 à 10 ans ?

De nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de tests et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les 5 à 10 ans à venir, que ce soit pour faciliter la vie des concepteurs de logiciel, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique, mais également pour prendre en compte de nouveaux enjeux de société tels que le développement durable et les économies d'énergie.

L'appel à défis pour les journées du GDR GPL à Pau en mars 2010, à sa suite, l'envoi du questionnaire de l'Institut INS2I vers les membres du conseil scientifique et les groupes de travail, et finalement la journée à Paris en juillet 2010 ont permis de collecter les questions sur lesquelles les équipes envisagent de travailler dans les années à venir. Les réponses à cet appel sont le fruit de discussions menées dans les groupes de travail et de contributions individuelles. A partir de ce matériel, nous dressons ici un panorama en 6 grandes familles thématiques, à savoir la structuration des logiciels pour leur évolution, la prise en compte de l'utilisateur final, l'accès aux composants et services, la prise en compte de la consommation de ressources dans les outils du logiciel, la modélisation de connaissances et de l'incertain et, finalement, la place des méthodes et outils de vérification et de validation.

Sans être exhaustif, ce texte regroupe, à notre avis, une grande partie des thèmes de la programmation et du génie logiciel abordés par les équipes françaises. Nous présentons chacune de ces 6 familles, puis nous donnons un ensemble de défis transverses qui sont apparus lors des échanges pendant la journée parisienne en juillet.

2. Structuration des logiciels pour leur évolution

2.1. *Concepts pour préparer un logiciel à évoluer – Jean-Marc Jézequel*

Les systèmes informatiques deviennent de plus en plus complexes. Cette complexité augmente de façon exponentielle avec un facteur dix tous les dix ans, ceci depuis plus de quarante ans. En dehors de toute considération académique ou industrielle, le fait que le développement d'un logiciel commence par une spécification bien définie et que le code soit écrit *from scratch* n'est plus vrai. Dans le monde réel de la programmation, le changement et l'évolution en continu sont la norme. En lieu et place de produire un logiciel pour un problème donné, l'objectif réel maintenant en génie logiciel est de produire des familles de logiciels évolutifs, soit dans l'une ou l'autre dimension, ou les deux. Ces deux dimensions sont le temps (versions successives) et l'espace (variantes d'un produit). En fonction du contexte, la prise en considération du délai de mise sur le marché, du coût et de la qualité rentrera en jeu dans la technique choisie pour la mise en œuvre de l'évolution de ce logiciel.

Un besoin croissant de concepts se fait sentir pour la construction de programme à partir de préoccupations développées indépendamment les unes des autres et pouvant être combinées de façon flexible, ceci en vue de faciliter la gestion d'évolution. Ces concepts doivent venir en complément des éléments structurants disponibles dans les langages actuels : modules, classes, fonctions, paquets, etc. Quelles que soient les structures choisies pour définir un système complexe, certaines préoccupations ne se composent pas si facilement, avec des méthodes de conception traditionnelle. Ces préoccupations peuvent correspondre à des moyens de synchronisation, de gestion de mémoire, des politiques de cache, du monitoring, etc. Elles sont connues comme des préoccupations transverses, parce que leur prise en compte doit être mise en place dans des petites portions de code réparties partout dans un programme correctement structuré, ce qui rend leur évolution coûteuse, pénible et risquée.

2.2. *Remodularisation des logiciels : lutter contre l'érosion de la structure et préparer la migration [Anquetil et al. 2010]*

Les systèmes orientés objet sont des modèles du monde réel qui manipulent une représentation de ses entités au travers de modèles de processus. Le monde réel n'est pas statique, de nouvelles lois sont créées, des concurrents offrent de nouvelles fonctionnalités, les utilisateurs ont de nouveaux besoins, des contraintes physiques peuvent s'ajouter. Un logiciel doit alors s'adapter continuellement, au risque, dans le cas contraire, de devenir rapidement obsolète. Dans le même temps, les applications sont devenues complexes et très grandes. Au-delà de la maintenance, une bonne structure permet d'avoir des systèmes logiciels de bonne qualité pour la migration vers des paradigmes modernes tels que les web services ou les composants et le problème de l'extraction de l'architecture est très proche du problème de remodularisation. Les lois de l'évolution du logiciel définies par Lehman et Belady

indiquent que les changements sont continus, c'est-à-dire qu'un logiciel qui résout un problème doit être continuellement adapté, sinon, il devient progressivement moins satisfaisant, et que plus un programme évolue, plus sa complexité augmente, alors que moins de travail est fait pour le maintenir ou le réduire.

Différentes solutions ont été étudiées pour construire des abstractions modulaires, pour proposer des approches de remodularisation, pour définir des techniques de modules. Aucune n'est pour le moment satisfaisante. Aussi plusieurs challenges se présentent. Le premier concerne une meilleure maîtrise de l'érosion et la préparation de la migration vers des technologies avancées. Le second devrait permettre la définition de bonnes abstractions et de leurs relations entre-elles, de la proposition d'algorithmes de remodularisation complémentaires en termes d'analyses, de complexité et d'approximation, mais également ces algorithmes devront passer à l'échelle.

Il est donc urgent de conduire des études complètes dans ce contexte de remodularisation, aussi bien verticalement par l'étude de tous les aspects du problème de modularisation (modélisation du logiciel, définition de métriques de qualité de modularisation, présentation des résultats) et horizontalement en considérant les différentes approches de modularisation. La solution sera sans doute multiple, prenant en compte différentes compétences dans différents domaines de recherche.

3. Prise en compte de l'utilisateur final

3.1. *End-user modelling (Albert et al. 2010)*

D'après le *Standish group*, l'amélioration du taux de réussite des projets informatiques dépend, entre autres, de l'implication des utilisateurs finaux. L'ingénierie logicielle a intégré depuis quelque temps dans ses approches, le fait d'associer plus étroitement l'utilisateur futur dans les phases de conception. Les démarches agiles, officialisées en 2001, ont popularisé ce principe. En parallèle, la production logicielle et la taille des applications informatiques ont accentué la nécessité de réutiliser les éléments logiciels. Une réponse est d'accorder plus d'importance à la modélisation des logiciels et d'y faire participer l'utilisateur final. Cette modélisation doit être le vecteur pour une discussion et une compréhension entre utilisateurs et concepteurs.

L'objectif de ce défi est de proposer aux experts d'un domaine les moyens de construire leur propre système informatique en utilisant des techniques de modélisation avancée telles que l'ingénierie dirigée par les modèles. Il s'agit ici de proposer une approche de la conception, non pas dirigée par la syntaxe d'un langage, mais par les concepts, les usages et les contraintes du métier. Les enjeux sont une productivité plus grande et une meilleure adéquation des produits aux problèmes. Ces enjeux ne seront atteints que si les artefacts de modélisation donnés sont adéquats et si les produits résultants sont fiables et opérationnels.

Les verrous identifiés sont des difficultés en termes de représentations à donner aux multiples profils des utilisateurs à mettre en adéquation avec leurs usages, au niveau de l'interprétation de ces notations par le système pour assurer l'aide, la validation, la collaboration et l'exécution des systèmes produits, et finalement l'impact sur les démarches usuelles de développement.

Les jalons de ce défi sont la circonscription de la complexité par des études de cas, la définition des artefacts de modélisation cognitivement adaptés aux différents rôles impliqués, la corrélation entre modélisation métier et mise en œuvre au niveau des plates-formes.

3.2. *L'évaporation des langages (Consel 2010)*

L'approche par DSL (*Domain Specific Language*) est utilisée depuis longtemps avec succès dans deux domaines historiques que sont la téléphonie et le développement d'applications Web. De l'ingénierie du logiciel aux langages de programmation, il y a un sentiment partagé qu'il reste encore beaucoup de travail pour que l'approche DSL soit un succès.

A la différence de langages de programmation généralistes qui ciblent des programmeurs entrainés, un DSL tourne autour d'un domaine : il vient d'un domaine et cible les membres de ce domaine. Ainsi, un DSL réussi doit être une sorte de langage qui se fait oublier. En quelque sorte, la programmation étend sa portée aux utilisateurs finaux, ce qui inclut les utilisateurs pour qui écrire des programmes vient en support de leur métier premier. Les exemples bien connus sont Excel ou encore MatLab.

Créer un langage qui sait se faire oublier repose conjointement sur une analyse de domaine et sur la conception de langage. Les expériences pratiques montrent que ces deux phases sont consommatrices de temps et à hauts risques. Comment ces deux phases doivent-elle être outillées ? Quelles améliorations pouvons nous en attendre ?

Un DSL réussi est surtout un DSL utilisé. Pour atteindre ce but, le concepteur doit avoir besoin de réduire, de simplifier, et de personnaliser un langage. Pour faire cela, le développement d'un DSL contraste avec la recherche dans les langages de programmation pour lesquels la généricité, l'expressivité et la puissance doivent caractériser tout nouveau langage. Une conséquence de ceci est que les experts en langage de programmation ne sont pas nécessairement les bonnes personnes pour développer un DSL. Est-ce que cela veut dire que pour un domaine donné, ses membres devraient développer leur propre DSL ? Ou qu'il devrait exister une nouvelle communauté d'ingénieurs langage qui font le lien entre les experts entre les experts en langage de programmation et les membres d'un domaine ? La communauté de recherche dans les langages de programmation s'est toujours très largement intéressée à la conception et à la réalisation des langages. Très peu d'effort ont été portés sur la compréhension de la façon dont les humains utilisent les langages. Cette piste de recherche est une clé pour étendre le domaine de la programmation au delà des

chercheurs en informatique. Comment devons-nous concevoir les langages pour qu'ils soient facilement utilisables ? Comment mesurons-nous le bénéfice de la productivité avec l'utilisation d'un langage ? Comment mesurons-nous la qualité des logiciels venant de programmes issus de DSL ?

Finalement un DSL est souvent une version simplifiée d'un langage de programmation généraliste : les constructions syntaxiques sont personnalisées, la sémantique est simple, les propriétés vérifiables par construction également. Ces différences clés peuvent provenir du manque d'outils pour le développement de DSL. Il y a aussi beaucoup d'outils de manipulation de programmes (générateurs de *parsers*, éditeurs, IDE) qui peuvent facilement être personnalisés pour de nouveaux langages, qu'ils soient graphiques ou textuels. De plus, pour une large classe de DSL, le compilation consiste à produire un code au-dessus d'un cadre de programmation spécifique à un domaine, et à permettre l'utilisation d'outils de transformation de haut niveau. Finalement, les propriétés peuvent souvent être vérifiées par des outils de vérification générique. Alors, que manque-t-il pour développer des DSL ? A-t-on besoin d'un environnement intégré pour le développement de DSL, orchestrant une librairie d'outils ? Devrait-il exister un nouveau type de compilateur et de générateur de vérification en relation avec les propriétés des DSL ?

4. Accès aux composants et services

4.1. *Fermes de composants et de services (Arévalo et al. 2010)*

Les composants logiciels et les web services sont des blocs de code qui sont utilisés dans la composition de logiciels modernes. Ils fournissent tous les deux des fonctionnalités qui demandent à être enregistrées dans des bases de données de façon à être accessibles et réutilisées dans des processus de construction de logiciels. Assembler des composants logiciels pour construire de nouveaux logiciels ou pour réparer ou faire évoluer des logiciels existants demande à ce que l'on puisse sélectionner le composant ou le service qui fournit une partie de la fonctionnalité applicative voulue et de le connecter facilement (avec un minimum d'adaptation) aux autres composants sélectionnés. De nombreux composants existent déjà. Par exemple, le moteur de recherche Seekda de web services contient plus de 28 000 références et le consortium OW2 regroupe plus de 40 projets open source dans le domaine des intergiciels. Cependant les modèles d'accès à ces références ne sont pas pour le moment satisfaisants. Il est alors nécessaire de concevoir des bureaux d'enregistrement (*registries*) pour collecter les composants et assister les utilisateurs dans leur recherche, leur sélection, leur adaptation et leur connexion d'un composant à d'autres. Nous imaginons que ces tâches pourraient être automatisées le plus possible et intégrées dans des environnements ouverts et dans des contextes dynamiques (périphériques embarqués administrés à distance, intelligence ambiante, applications ouvertes et extensibles, informatique mobile, etc.).

Le défi majeur consiste à proposer une architecture pour un bureau d'enregistrement des composants et des services, disponible en ligne, qui ne donnera pas uniquement un accès aux composants adéquats, mais proposera également un support tout au long du cycle de vie aux développeurs et aux applications (si possible en mode automatique). Ce bureau d'enregistrement sera une plate-forme de partage de connaissances et d'expériences sur les composants en direction des développeurs. Celle-ci leur permettra aussi bien de tester leurs développements efficacement que de les exécuter dans un environnement fiable. Une extension de ce moteur de recherche est naturellement une ferme de composants où les composants pourront être soit localisés physiquement (modèle de dépôt de composants) ou uniquement référencés dans un répertoire de composants organisé de manière adéquat. Cette ferme de composants pourra offrir des vues multiples sur les composants et proposer des mécanismes de recherche efficaces.

4.2. *Software as a Service (SaaS) et Cloud Computing – Lionel Seinturier*

L'approche service et l'informatique dans les nuages sont deux tendances actuelles fortes du développement logiciel. Au delà de l'attente suscitée à tort ou à raison, il s'agit d'une remise en cause profonde de la façon dont les logiciels sont développés et hébergés et du modèle économique qui sous-tend leur utilisation. L'approche SaaS peut être définie simplement comme un logiciel déployé comme un service hébergé et accédé via Internet. Les gains attendus d'une telle approche résident dans l'accès universel et ubiquitaire au logiciel, les gains en terme de maintenance et de mise à jour des infrastructures matérielles et système la tolérance aux pannes et le passage à l'échelle.

Les défis impliqués par l'approche service et l'informatique dans les nuages sont nombreux tant en terme de développement logiciel qu'en termes d'usage, confidentialité et propriété des données, mode économique de facturation. De part la nature distribuée des traitements et des données qui sont mises en jeu, ces défis concernent plusieurs communautés dont celle du génie logiciel (GDR GPL), mais aussi celles des systèmes distribués (GDR ASR) et des systèmes d'information (GDR I3). En terme de génie logiciel, un défi concerne la définition d'un modèle de services et composants universel et standardisé qui supporte le cycle de développement complet du SaaS, de la conception, à l'implémentation, au packaging, au déploiement, à l'exécution, au monitoring, à la reconfiguration. Ce modèle devra supporter un degré élevé d'hétérogénéité en terme de langages de description des services et de langages de programmation. Les descriptions de services devront pouvoir être enrichies avec des informations de niveau sémantique et non fonctionnelles liées par exemple à leur consommation énergétique. Ce modèle de services devra pouvoir être mis en œuvre aussi bien pour les applications SaaS que pour le développement des plates-formes de Cloud Computing supportant ces applications. En terme de plate-forme, il s'agira de faire en sorte de proposer différents niveaux de virtualisation des ressources logicielles et matérielles afin d'assurer une cohabitation harmonieuse, fiable et sûre des différents services hébergés. L'architecture de ces plates-formes devra être scalable afin de pouvoir envisager des usages de style Home Cloud pour l'hébergement de services

dans des domaines tels que l'immotique et la domotique, sur des équipements tels que des set-top box, les compteurs intelligents ou les boitiers de raccordement Internet. En conclusion, ce défi consiste à faire émerger un modèle viable et fédérateur pour l'industrie du logiciel.

4.3. *Internet des ressources : Connecter le yocto physique au yotta virtuel – Romain Rouvoy*

Internet ne cesse de se développer afin d'offrir toujours plus de fidélité vis-à-vis du monde physique. Si l'émergence du Web 2.0 a offert de nouveaux outils collaboratifs pour diffuser un contenu plus dynamique et faciliter l'expression des internautes au travers des applications Internet riches (blog, forums, wiki), le monde physique demeure partiellement connecté au monde virtuel. La prochaine évolution de l'Internet se devra donc d'intégrer plus facilement toutes les dimensions du monde physique pour offrir aux internautes de nouvelles expériences. Dans cette perspective, on observe une mouvance croissante autour du concept de *ressource logicielle* car il semble offrir une abstraction adéquate pour connecter les objets du quotidien, du plus petit au plus grand. À l'avenir, l'internaute pourra ainsi naturellement rechercher et publier des services virtuels (co-voiturage) ou physiques (station météo) en les exposant sous la forme de ressources logicielles sur Internet. Du point de vue des usages, la prise en compte de la sécurité (authentification, encryptage) et du respect de la vie privée (diffusion aux membres du réseau social, diffusion sous anonymat) représente une dimension importante du succès de cet Internet démocratisé. D'un point de vue technique, cette intégration est une tâche complexe qui nécessite de prendre en compte la diversité matérielle et logicielle des objets communicants (protocoles de communication, de découverte, d'interaction). Dès lors, il apparaît critique de reconsidérer les méthodes et les outils d'ingénierie logicielle permettant de décrire, concevoir, configurer, et composer ces ressources logicielles afin d'offrir aux utilisateurs des interfaces flexibles et intuitives pour contrôler la diffusion et l'accès du public à leurs objets. Ainsi, au travers de cette mise en relation des ressources distribuées sur l'Internet, il sera possible à terme de déployer des infrastructures collaboratives à très large échelle permettant aux internautes d'interagir à distance via les objets de leur quotidien.

5. Modèles de développement pour environnements contraints

5.1. *De la réification de l'énergie dans le domaine du logiciel (Menaud et al. 2010)*

En quelques années, le problème de la gestion de l'énergie est devenu un enjeu de société. En informatique, les principaux travaux se sont concentrés sur des mécanismes permettant de maîtriser l'énergie au niveau du matériel. Le renforcement du rôle de l'informatique dans notre société (développement des centres de données, prolifération des objets numériques du quotidien) conduit à traiter ces problèmes aussi au niveau du logiciel. Nous nous posons la question de la

réification de l'énergie comme fut posée en son temps celle de la réification de la mémoire (l'espace) et de l'interpréteur (la machine d'exécution).

Le défi est ici de sensibiliser dans un premier temps l'utilisateur final au problème de la consommation énergétique en lui offrant des mécanismes d'introspection visualisant la ressource énergie à l'image de ce qui se fait aujourd'hui dans le domaine automobile (consommation d'essence instantanée). Il s'agira ensuite de proposer des mécanismes d'intercession aux développeurs aptes à contrôler cette consommation énergétique. Ces mécanismes réflexifs devront concerner l'ensemble du cycle de vie du logiciel.

5.2. *Programmation pour le calcul haute performance - Gaétan Hains*

L'adaptation des modèles et outils de programmation parallèle aux architectures hétérogènes GPU-multicoeur-multiprocesseur-cloud et l'intégration de notions de très haut niveau pour la gestion des ressources de calcul: cpu/temps/espace/KW/€. Un thème émergent est celui de la fusion de ces travaux "programmation" avec les technologies industrielles du monitoring (métrologie) des centres de calcul, celles-ci étant essentiellement au niveau système et pas encore middleware.

5.3. *Adaptation diffuse des logiciels –Sylvie Vignes (Vignes 2010)*

Un environnement diffus peut se composer d'objets hétérogènes, tels que des objets émettant et recevant des informations via des réseaux de capteurs ou d'autres éléments logiciels. Le fait de positionner un logiciel dans un environnement de ce type et être capable d'interagir avec celui-ci nécessite de prendre en compte à la fois l'environnement et le logiciel dans leur globalité, et ceci dans toutes les étapes du cycle de vie du logiciel. Les premières étapes, c'est-à-dire l'analyse des besoins, doit être revue pour permettre l'expression à la fois du comportement global, mais également des comportements individuels. Les phases de conception, de développement et d'exécution doivent être également revisitées pour proposer des méthodes et outils permettant la prise en compte des interactions avec l'environnement du logiciel. Ce travail sur l'élaboration de nouvelles approches de développement pour des environnements diffus doit permettre la prise en compte de l'incertain tel que décrit dans la section 6 et offrir des capacités d'évolution tel que décrit dans la section 2.

6. Modélisation de connaissances et de l'incertain

6.1. *Identifier et intégrer les connaissances des développeurs dans les outils de développement – Mikal Ziane*

Pourquoi est-il si difficile de développer un logiciel ? Peut on être satisfait de l'état actuel du génie logiciel ? Imaginer qu'un non informaticien puisse un jour assembler lui-même une application simple sur son ordinateur pour créer un nouveau programme est-ce trop demander ? Pourquoi les logiciels sont-ils toujours si couteux à créer et maintenir et si peu fiables ?

Les outils sont ignares. Les outils actuels d'aide au développement sont typiquement ignares en ce sens qu'ils ne partagent que très peu de connaissances avec les développeurs sur la manière de développer du logiciel (et *a fortiori* sur les domaines d'application des logiciels qu'ils aident à développer). La conséquence est que le dialogue entre les développeurs et leurs outils se situe trop souvent à un très bas niveau d'abstraction, au niveau du code, ce qui limite drastiquement les possibilités d'utilisation de ces outils. Ce constat a déjà été fait dans les années 1980 voire avant mais il est toujours d'actualité.

Une thématique a disparu. Dans les années 80 et jusqu'au milieu des années 1990 le génie logiciel à base de connaissances était un thème de recherche reconnu [KBSE 1990], à l'intersection du génie logiciel et de l'intelligence artificielle. Son objectif était d'identifier les connaissances sous-jacentes au développement au logiciel et à les intégrer dans des outils. Ce n'est plus le cas. Ainsi la conférence *Knowledge-Based Sofware Engineering* s'est elle renommée *Automated Software Engineering* en 1997. Pourquoi ?

Sans doute la tâche était-elle trop ambitieuse à l'époque. Pourtant des tentatives qui n'affichent pas forcément explicitement cette thématique vont de temps à autre dans la même direction. Citons par exemple la tentative de Microsoft avec le projet *Intentional Programming* [IP 1996]. Remarquons aussi que l'initiative MDA de l'OMG [MDA] visait à séparer « *business and application logic from the underlying platform technology* » sans d'ailleurs s'en donner les moyens scientifiques.

La situation est plus favorable aujourd'hui qu'il y a 20 ans. Un ensemble de signes suggèrent qu'aujourd'hui pourrait être le bon moment pour à nouveau, explicitement, chercher à identifier et à intégrer à des outils, non seulement les connaissances propres au développement logiciel mais aussi les connaissances des domaines applicatifs :

– Les outils de transformations ou de manipulation de programmes sont plus mûrs (TOM, Stratego, ASF+SDF/Rascal, ANTLR, etc.). Or les connaissances de développement sont en premier lieu des transformations.

– Le développement de langages de domaines (DSL) est en plein essor, parfois adossés à des ontologies, bien que combler le hiatus entre les deux reste un problème ouvert.

– Des plates-formes comme Eclipse facilitent l'intégration d'outils dans un ensemble relativement cohérent.

– Le développement de technologies à base de composants, les intergiciels, les architectures à bases de services, même s'ils posent de nouveaux problèmes, facilitent globalement le développement du logiciel : il ne s'agit plus nécessairement de synthétiser un programme mais d'assembler et déployer des éléments de solution.

Quels défis pour demain ? Vu la difficulté de la tâche (à la mesure de son ambition) il semble qu'il faut définir les défis sur cette thématique selon des critères suivants :

– Spécifiques au domaine : on sait que des connaissances de domaine sont nécessaires. Il faut se focaliser sur des domaines abordables pour lesquels on sait déjà faire du logiciel mais pas un domaine trivial,

– Connaissances opérationnelles : il ne s'agit pas de tomber dans l'écueil d'ontologies de domaines totalement inutilisables par un outil d'aide au développement,

– conduit par les tests : il faut un moyen automatique de vérifier/spécifier (partiellement) que le logiciel produit fonctionne. Pas directement une spécification formelle car un utilisateur ne saura pas la produire, plutôt des tests partiels,

– Connaissances explicites : les connaissances utilisées pour automatiser partiellement telle ou telle étape du cycle de vie doivent absolument être explicitées et on doit pouvoir raisonner dessus. Ceci implique qu'elles soient formalisées rigoureusement. C'est le défi mais c'est aussi le levier. Si on sait un peu exprimer des transformations de programmes, on raisonne mal dessus et on sait mal exprimer le contrôle de ces transformations.

6.2. *Auto-adaptation des logiciels, l'incertitude en tant qu'intrus dans le cycle de vie – Laurence Duchien*

La volonté d'accéder à l'information en continu et le fait que les technologies soient de plus en plus intégrées dans nos lieux de vie invitent à repenser la manière dont nous construisons, gérons et exécutons les logiciels. Pour que les logiciels deviennent flexibles, configurables ou reconfigurables et paramétrables tout en étant résistants et fiables, il est nécessaire de revisiter leur cycle de vie complet. En effet, ceux-ci doivent s'adapter en prenant en compte les changements de leurs contextes opérationnels, de leurs environnements ou encore de leurs propres caractéristiques. Dès lors, l'auto-adaptation devient un élément prépondérant des différentes phases de vie des logiciels. Même si ce sujet est déjà traité dans le cas de la mise en place de plates-formes d'exécution ad-hoc dans de nombreux domaines tels que les systèmes autonomes, les systèmes embarqués, les réseaux ad hoc ou mobiles, la robotique, l'intelligence ambiante, les réseaux de capteurs ou encore les architectures orientées services, il est nécessaire de visiter les différentes phases de conception des applications associées, c'est-à-dire la prise en compte de l'évolution à tout moment.

La gestion de cette fonctionnalité d'auto-adaptation reste un défi majeur dans le domaine de l'ingénierie du logiciel, ceci malgré les avancées récentes dans les différents domaines cités ci-dessus. Les défis à relever concernent essentiellement la représentation de l'incertitude dans la modélisation, l'expression des besoins, l'ingénierie et la validation de ces systèmes. En modélisation, différentes dimensions devront permettre la représentation des propriétés associées au raisonnement ayant lieu à l'exécution et la prise de décision pour mettre en œuvre l'auto-adaptation. Il s'agit par exemple de l'évolution, de la flexibilité, de la multiplicité des évolutions, de la dépendance de ces évolutions ou encore de leur fiabilité. On représentera alors la fréquence des changements, leur anticipation et leur impact. Les effets de ces changements pourront être définis en termes de criticité, de prédictibilité, *d'overhead* ou encore de résilience. Le défi pour l'expression des besoins concerne la définition de nouveaux langages capables de représenter l'incertain à un haut niveau d'abstraction et de le gérer. Au niveau de l'ingénierie, la boucle de contrôle devient une entité de première classe tout au long du processus de développement et d'exécution. Pour cela, il est nécessaire de modéliser le support qui permet la réalisation de cette boucle et de prendre ce modèle en compte dans la réalisation des applications. Cela devrait permettre une réification des propriétés du système et faciliter le raisonnement sur les propriétés du système. Finalement, la validation des systèmes auto-adaptables passe par l'identification dynamique des nouveaux contextes et donc la prise en compte de l'incertitude que ce soit en termes de modèles et de leur validation.

7. Méthodes et outils de vérification et validation

7.1. *Langages, Types et Preuves (Castéran, Dubois 2010)*

Le groupe LTP (Langages, Types et Preuves) [Castéran, Dubois 2010] regrette la diffusion limitée des formalismes, langages et outils permettant de produire des logiciels sûrs, malgré leur existence en nombre significatif et s'interroge sur leur diffusion limitée. Sa première proposition est de faire passer les preuves dans le quotidien du développeur en permettant aux techniques existantes de passer à l'échelle. Pour cela, il préconise d'automatiser le plus possible les preuves en utilisant des prouveurs automatiques, de faire collaborer les techniques (test, model checking, analyse statique et preuves) qui sont complémentaires et, finalement, de réutiliser des preuves existantes pour passer à l'échelle. La seconde proposition concerne la rupture entre le semi-formel et le formel qui est, à son avis, trop grande. La réduction de cette rupture est possible si l'on restreint les formalismes utilisés dans les outils tels que B ou Coq à des domaines d'application où un langage spécifique facilite la lecture des spécifications tout en étant accompagné d'une sémantique formelle très précise. Cette approche permettrait une jonction entre DSL (langages dédiés à un domaine) (voir section 3) et les outils de preuve formelle.

7.2. *Méthodes Formelles pour le Développement Logiciel (Potet, Aït Ameur, Laleau 2010)*

Le groupe MFDL (Méthodes Formelles pour le Développement Logiciel) [Potet, aït Ameur, Laleau 2010] s'intéresse une intégration maîtrisée et raisonnée des méthodes formelles dans un cycle de développement de systèmes à logiciel prépondérant. Il est alors envisageable d'appliquer différentes méthodes et approches de modélisation et de vérification dans un continuum, de l'analyse du système à la production du code sur un matériel spécifique, de bout en bout. Une traçabilité formelle entre les différents niveaux du cycle de vie d'un processus est également l'un des défis à relever. Cela devrait permettre une production de logiciels certifiés. Comme pour le groupe LTP, le groupe MFDL souligne également l'importance du passage à l'échelle dans l'utilisation des méthodes formelles, ici dans les processus industriels et l'importance d'un meilleur enseignement de ces approches aux futurs ingénieurs.

7.3. *Méthodes de Test pour la Validation et la Vérification (Gotlieb, Groz 2010)*

Finalement, le groupe MTV2 (Méthodes de Test pour la Validation et la Vérification) [Gotlieb, Groz 2010] souligne que la complexité des systèmes à logiciel prépondérant demande que les processus et techniques de tests soient mieux définis, fiables et capables de passer à l'échelle. Trois défis sont proposés par ce groupe. Le premier consiste à mieux intégrer les tests dans les méthodes de développement, ceci dès les phases de modélisation et de manière continue et traçable jusqu'à la production du code. La co-évolution modèle-code est également un aspect à prendre en compte dans la génération de tests. Le second défi consiste à réconcilier test et fiabilité. La grande dépendance de fonctionnement des entreprises, et plus généralement de la société, vis-à-vis des logiciels rend nécessaire une évaluation quantitative de leur fiabilité. Les procédures de tests reposent sur l'évaluation des défauts de fonctionnement. Il est alors nécessaire d'établir un lien fort entre test et fiabilité du logiciel en étudiant les notions de couverture liées à des profils d'utilisation et d'évaluer des probabilités d'événements rares à partir de techniques de simulation. Le dernier défi consiste à prendre en compte les environnements nouveaux d'exécution, tels que les systèmes mobiles ou ubiquitaires, ou encore des systèmes hybrides. Les caractéristiques de ces environnements posent de nouveaux défis en matière de tests et demandent à être étudiées. Finalement, comme pour les deux groupes précédents, le groupe MTV2 pose le problème de l'enseignement du test, et de la transformation du métier d'ingénieur informatique en France. En effet, ce métier, avec l'apparition des développements off-shore, est maintenant amené à contrôler plus avant la qualité du logiciel développé en dehors de nos frontières, et donc à maîtriser les techniques de test.

8. Une analyse transverse des 7 familles thématiques

A la suite des présentations et des discussions qui ont eu lieu à Pau en Mars 2010 et à Paris en juillet 2010, plusieurs thèmes non identifiés en tant que tels sont revenus dans plus d'une présentation et ont suscité des débats. Nous en donnons ici une liste.

– Continuum dans le développement : les frontières entre différentes phases du développement du logiciel sont devenues floues. On va vers un continuum qui s'étend jusqu'à l'exécution du logiciel et à sa maintenance. Ce continuum va permettre d'enrichir de façon continue le logiciel et devra s'appuyer sur une traçabilité forte de l'analyse jusqu'à la conception du logiciel. Il est également à remarquer que l'introduction des propriétés extra-fonctionnelles devrait se faire au plus tôt dans le cycle de développement.

– Approches agiles : le développement du logiciel par des méthodes agiles avec des cycles de développement courts. La notion de méthode agile s'étend au-delà de la programmation. Ce type de méthode s'appliquerait également dans les différentes étapes du cycle de vie du logiciel. Il s'agit plus d'une méthode alliant la rapidité et une intégration à la fois continue et fiable des différents éléments constitutifs du logiciel.

– Connaissances et développement : l'appui sur des connaissances issues des réseaux sociaux (Web 2.0), mais également des forums, mails, etc. devrait apporter une aide à la fois lors du développement et lors de l'évolution du logiciel. En effet, il s'agit ici de constituer les documentations, le suivi du logiciel, mais également une aide à la conception par les informations échangées entre les développeurs, mais également par les utilisateurs. Ces connaissances ont besoin d'être structurées, traitées et enrichies de façon continue.

– Usages et utilisateurs : l'utilisateur est au centre du dispositif du cycle de vie. Il est important de dialoguer avec lui tout au long de la vie du logiciel. Ce défi rejoint la notion de méthode agile. Il s'agit de proposer un dialogue constant dans les phases d'élaboration du logiciel avec l'utilisateur final. Une autre voie d'échange avec les utilisateurs finaux est l'apprentissage des usages du logiciel et son optimisation via des approches expérimentales fondées sur des traces.

– Développement et exécution ouverts : les environnements de développement et d'exécution deviennent plus ouverts. Ils sont évolutifs jusqu'à aller vers l'autonomie. Par autonomie, on pense à la supervision, mais également à l'auto-gestion avec l'auto-optimisation, l'auto-protection, l'auto-guérison, l'auto-test ou encore l'auto-adaptation.

– Complexité et passage à l'échelle dans l'utilisation des approches formelles : dans l'utilisation des méthodes formelles, que ce soit pour vérifier des propriétés, pour accompagner des processus de développement ou encore pour tester des parties logicielles, il devient important de prendre en compte la complexité des logiciels actuels et de permettre le passage à l'échelle.

– Remodularisation ou langage pivot : il semble important d'avoir des vues sur des objets existants, quelque soit l'étape du cycle de vie et de pouvoir manipuler ces

objets selon cette vue, aussi bien pour faire évoluer le logiciel que pour l'analyser. Par exemple, avec les outils de preuve tels que Coq ou B, il serait intéressant de réaliser des vues dynamiques.

– Contractualisation et responsabilité face aux utilisateurs du logiciel : les concepteurs des logiciels devraient être conscients des enjeux des logiciels qu'ils développent. Une forme de contractualisation est à trouver vis-à-vis des utilisateurs pour améliorer la qualité des logiciels, mais aussi pour responsabiliser les concepteurs de logiciels.

9. Conclusion

La maîtrise du développement de logiciel est une technologie clé pour l'innovation et la performance économique d'un pays. Depuis sa création, l'informatique n'a pas cessé d'évoluer et de se révolutionner, tant dans le domaine du matériel que des paradigmes logiciels. Cette révolution permanente a permis à l'informatique de devenir moteur de progrès et d'innovation dans tous les domaines de l'activité humaine.

L'objet d'étude de la Science Informatique est un objet en mouvement, qui lève de nouveaux défis à chaque révolution, et remet en cause les solutions apportées aux défis du passé. Ces défis requièrent des avancées tant sur le plan fondamental que dans la mise en œuvre de résultats de la Recherche.

Dans ce document, nous avons essayé d'identifier plusieurs défis que des équipes du GDR GPL souhaitent relever dans les prochaines années. Ce document n'a pas l'ambition d'être exhaustif, mais il nous semble représentatif de la diversité des compétences de la communauté française en GPL et de l'intérêt de soutenir ses recherches.

10. Bibliographie

[Anquetil et al. 2010] N. Anquetil, S. Denier, S. Ducasse, J. Laval, D. Pollet (INRIA), R. Ducournau, R. Giroudeau, M. Huchard, J.C. König et D. Seriai (LIRMM / CNRS / Univ. Montpellier 2), Software (re)modularization : Fight against the structure erosion and migration préparation, Session défis, Journées du GDR GPL, Pau 2010

[Arévalo et al. 2010] Gabriela Arévalo (LIFIA / UNLP _ Argentine), Zeina Azmeh, Marianne Huchard, Chouki Tibermacine (LIRMM / CNRS / Univ. Montpellier 2), Christelle Urtado et Sylvain Vauttier (LGI2P / Ecole des Mines d'Alès) Component and Service Farms, Session défis, Journées du GDR GPL, Pau 2010

[Albert et al. 2010] Patrick Albert (IBM), Mireille Blay-Fornarino (I3S), Philippe Collet (I3S), Benoit Combemale (IRISA), Sophie Dupuy-Chessa (LIG), Agnès Front (LIG), Anthony Grost (ATOS), Philippe Lahire (I3S), Xavier Le Pallec (LIFL), Lionel Ledrich (ALTEN Nord), Thierry Nodenot (LIUPPA), Anne-Marie Pinna-Dery (I3S), et Stéphane Rusinek (Psitec), End-User Modelling, Session défis, Journées du GDR GPL, Pau 2010.

[Castéran, Dubois 2010] Pierre Castéran, Catherine Dubois, Proposition de défis – Groupe LTP, Appel à défis, Juin 2010.

[Consel 2010] Charles Consel (INRIA / Univ. Bordeaux) Towards Disappearing Languages, Session défis, Journées du GDR GPL, Pau 2010.

[Gotlieb, Groz 2010] Arnaud Gotlieb, Roland Groz, Test Logiciel à Grande Echelle, synthèse du groupe de travail MTV2 en réponse à l'appel à défi GDR-GPL, Appel à défis, Juin 2010.

[KBSE 1990] Knowledge-Based Software Engineering, in Avron Barr (Author), Paul R. Cohen (Author), Edward A. Feigenbaum (Editor) The Handbook of Artificial Intelligence, Addison-Wesley, 1990, vol 4, chap. XX.

[IP 1996] Charles Simonyi, Intentional Programming - Innovation in the Legacy Age Presented at IFIP WG 2.1 meeting, June 4, 1996

[MDA] http://www.omg.org/mda/

[Menaud et al. 2010] Jean-Marc Menaud, Adrien Lèbre, Thomas Ledoux, Jacques Noyé Pierre Cointe, Rémi Douence et Mario Südholt (Ecole des Mines de Nantes / INRIA / LINA) Vers une réification de l'énergie dans le domaine du logiciel _ L'énergie comme ressource de première classe, Session défis, Journées du GDR GPL, Pau 2010.

[Potet, Aït Ameur, Laleau 2010] Marie-Laure Potet, Yamine Aït-Ameur, Régine Laleau, Intégration maîtrisée et raisonnée des modèles et méthodes formelles dans un processus industriel de développement logiciel, Appel à défis, Juin 2010.

[Vignes 2010] Sylvie Vignes, Contribution à l'Ingénierie des Besoins spécifique à l'adaptation diffuse, Appel à défis, Juin 2010.